Catching 

The (Baseline) Unicode Plan for C++23

ThePhD | @thephantomderp | phdofthehouse@gmail.com
https://linkedin.com/in/thephd | https://thephd.github.io | https://github.com/ThePhD

In the Year 2019…

- Unicode support in C and C++ is a grim tale
  - <cctype>
  - wstring_convert needed to be pulled for causing great harm to performance and more
  - Current conversion routines suck
  - Save a file in Germany with the default locale on Windows, ship it to Japan
    - Enjoy the hilarity

What makes encoding hard?

Pain Points

Pain: Various Flavors of Incompatibility

- Compile time: “`20 Å`“
  - such literals may not compile cleanly on all platforms;
  - or, it compiles with only warnings and then mangles the data

- Difficult to write programs which handle text portably at runtime
  - Locale-sensitivity creates enormous issues
    - Text written and serialized by iostreams in Czech republic does not read properly in Japan, or in the U.S., or…

Fix: Change default C locale to be UTF8

- “default”?
  - Yes – mandating is too strong
  - But, changing the default (with a way out) is more palatable
  - Encouragement from Axel Andrejs, Microsoft lead
  - Linux is already UTF8-by-default

- Vastly improves sharing of default character handling of C programs

- Do not expect overnight
  - 6+ year process

```
---
title: C.UTF8 - Standardizing for Portability
document: nXXX1
date: 2019-09-21
audience:
  - WG14
author:
  - name: JeanHeyd Meneide
    email: <phdofthehouse@gmail.com>
---
```

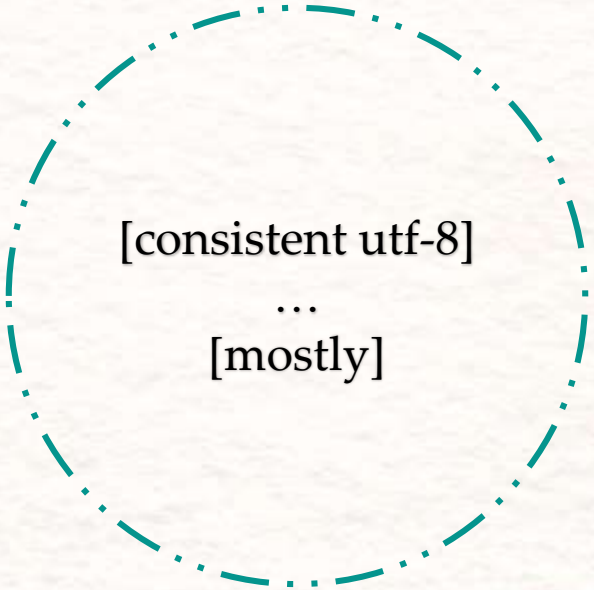
☣ Pain: Poisoned Input ☣

- Easy to end up with text nobody anticipated and encodings nobody knows
  - Encoding information is usually out-of-band (or non-existent)

- Internal state thrown into chaos and unpredictability
  - Symptom: ad-hoc checks sprinkled throughout the codebase
  - Symptom: mojibake floating through the system at random points

Pain: Permeable Boundaries

- Library/application author is internally consistent and aware of encoding
  - Still, receive text and did not know how to convert it properly
  - Ends up as mojibake, but is easier to find problems



Pain: Conversion Routines

- C has insufficient to/from encoding schemes and is not extensible
  - Unicode support is limited to one-by-one code point conversion functions
  - Neither performant, nor helpful

- What is the encoding of `char` and `wchar_t`?
  - You don't know, and it's very hard to tell
  - Entirely platform specific, sometimes even architecture dependent

Fix: More C functions

- Add:
  - String versions of slow, one-off character converting functions:
    - `c[8,16,32]s[r]tombs`, `mbs[r]toc[8,16,32]s`
    - `And only these`

- Tractable: full gamut is close to 40+ extra functions

```
title: Fast, Scalable String Support for Unicode
document: nXXX1
date: 2019-09-21
audience:
  - WG14
author:
  - name: JeanHeyd Meneide
    email: <phdofthehouse@gmail.com>
```

Multibyte (char*) string conversion functions:

```
size_t c8stombs(char* __dst, const char* __src, size_t __dst_len);
size_t c8srtombs(char* __dst, const char** __psrc, size_t __dst_len, mbstate_t* __ps);
size_t mbstoc8s(char* __dst, const char* __src, size_t len);
size_t mbsrtoc8s(char* __dst, const char** __psrc, size_t __dst_len, mbstate_t* __ps);

size_t c16stombs(char* __dst, const char16_t* __src, size_t __len);
size_t c16srtombs(char* __dst, const char16_t** __psrc, size_t __len, mbstate_t* __ps);
size_t mbstoc16s(char16_t* __dst, const char* __src, size_t __len);
size_t mbsrtoc16s(char16_t* __dst, const char** __psrc, size_t __len, mbstate_t* __ps);

size_t c32stombs(char* __dst, const char32_t* __src, size_t __len);
size_t c32srtombs(char* __dst, const char32_t** __psrc, size_t __len, mbstate_t* __ps);
size_t mbstoc32s(char32_t* __dst, const char* __src, size_t __len);
size_t mbsrtoc32s(char32_t* __dst, const char** __psrc, size_t __len, mbstate_t* __ps);
```

Wide (wchar_t) single-character conversion functions:

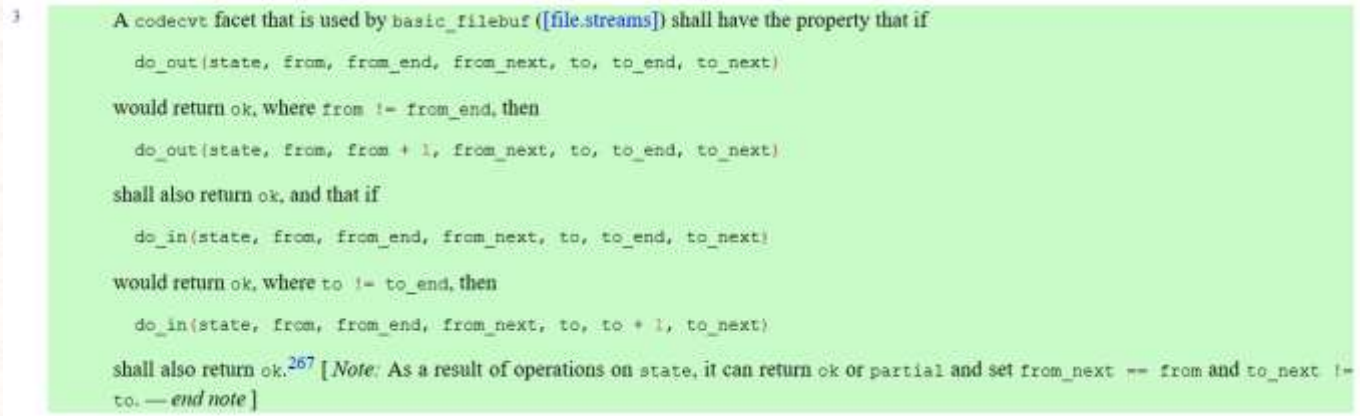
```
size_t c8rtowc(wchar_t* __dst, char __c8, mbstate_t* __ps);
size_t wcrtoc8(char* __dst, const wchar_t* __src, size_t __len, mbstate_t* __ps);
size_t c16rtowc(wchar_t* __dst, char16_t __c16, mbstate_t* __ps);
size_t wcrtoc16(char16_t* __dst, const wchar_t* __src, size_t __len, mbstate_t* __ps);
size_t c32rtowc(wchar_t* __dst, char32_t __c32, mbstate_t* __ps);
size_t wcrtoc32(char32_t* __dst, const wchar_t* __src, size_t __len, mbstate_t* __ps);
```

Wide (wchar_t*) string conversion functions:

```
size_t c8stowcs(wchar_t* __dst, const char* __src, size_t __dst_len);
size_t c8srtowcs(wchar_t* __dst, const char** __psrc, size_t __dst_len, mbstate_t* __ps);
size_t wcstoc8s(char* __dst, const wchar_t* __src, size_t __dst_len);
size_t wcsrtoc8s(char* __dst, const wchar_t** __psrc, size_t __dst_len, mbstate_t* __ps);

size_t c16stowcs(char* __dst, const char16_t* __src, size_t __len);
size_t c16srtowcs(char* __dst, const char16_t** __psrc, size_t __len, mbstate_t* __ps);
size_t wcstoc16s(char16_t* __dst, const char* __src, size_t __len);
size_t wcsrtoc16s(char16_t* __dst, const char** __psrc, size_t __len, mbstate_t* __ps);

size_t c32stowcs(wchar_t* __dst, const char32_t* __src, size_t __len);
size_t c32srtowcs(wchar_t* __dst, const char32_t** __psrc, size_t __len, mbstate_t*
__ps);
size_t wcstoc32s(char32_t* __dst, const wchar_t* __src, size_t __len);
size_t wcsrtoc32s(char32_t* __dst, const wchar_t** __psrc, size_t __len, mbstate_t*
__ps);
```

“At least C++ will—”

- No.
  - C++ is equally unhelpful, except for UTF-8 it uses `char8_t` (soon)
  - `wchar_t` still sucks here: `basic_filebuf` is restricted to 1:1 conversion for all input and output code units [locale.codecvt.virtuals#3]
  - `std::wstring_convert` failed miserably



3 A `codecvt` facet that is used by `basic_filebuf` ([file.streams]) shall have the property that if

```
do_out(state, from, from_end, from_next, to, to_end, to_next)
```

would return `ok`, where `from != from_end`, then

```
do_out(state, from, from + 1, from_next, to, to_end, to_next)
```

shall also return `ok`, and that if

```
do_in(state, from, from_end, from_next, to, to_end, to_next)
```

would return `ok`, where `to != to_end`, then

```
do_in(state, from, from_end, from_next, to, to + 1, to_next)
```

shall also return `ok`.[267] [*Note:* As a result of operations on `state`, it can return `ok` or `partial` and set `from_next == from` and `to_next != to`. — *end note*]

No Hotfix for <iostream>

- Get rid of so-called “1:1” rule?
  - Okay, but…

- Bigger problem:
  - wchar_t must be big enough to represent all codepoints (!)
  - picking ANYTHING except UTF32 means ruination
  - Therefore, even today, Windows is UCS-2 for almost all C and C++ wchar_t library functions (!!)

The C++ Fix

P1629
Standard Text Encoding

The Goal 🛣

- What we are aiming for:

```
#include <text>
#include <iostream>

int main () {
        using namespace std::literals;

        std::text::u8text my_text = std::text::transcode<std::text::utf8>("안녕하세요 👏"sv);
        std::cout << my_text << std::endl; // prints 안녕하세요 👏 to a capable console
        std::cout << std::hex;
        for (const auto& maybe_cp : my_text ) {
                std::cout << static_cast<uint32_t>(*maybe_cp) << " ";
        }
        // 0000c548 0000b155 0000d558 0000c138 0000c694 00000020 0001f44b
}
```

Keep this dream in mind…

Or: “How do we get to there, from here?”

Digging Deep

Foundation: encoding objects

- Encoding is a concept that a class can satisfy
  - It has some required (and optional) operations

- Serves as the foundational building block
  - Encodes and decodes one code point at a time
  - Member types and static member variables to dictate some useful defaults

An example encoding object

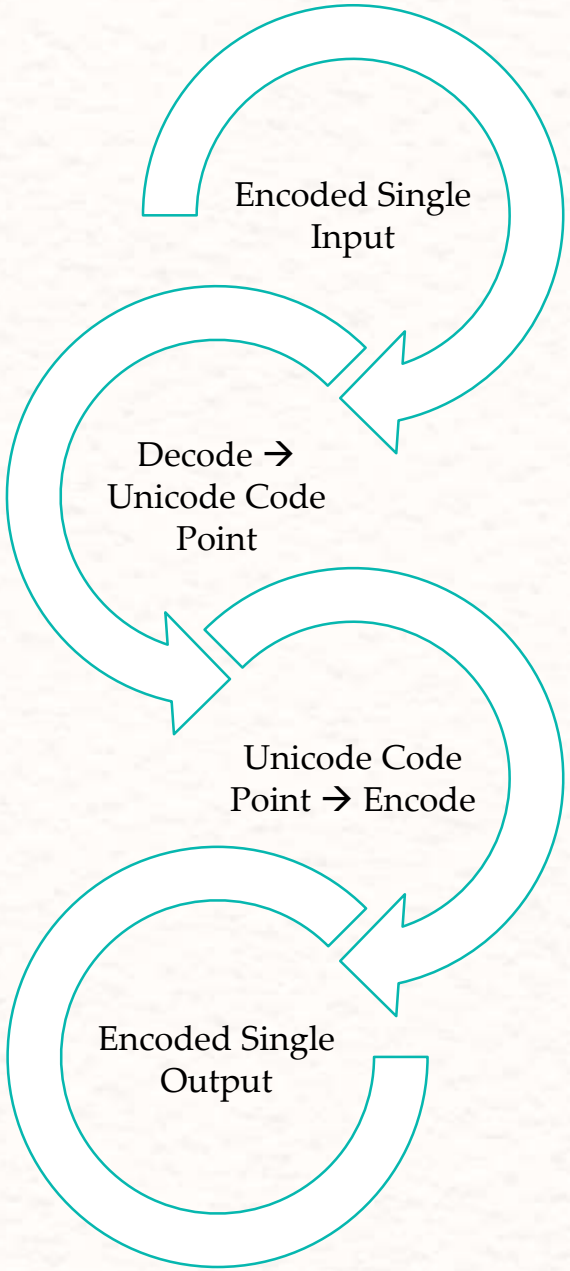
```
struct utf8 {

        using state               = __text_detail::__empty_state;
        using code_unit           = char8_t;
        using code_point          = unicode_code_point;
        using is_decode_injective = std::true_type;
        using is_encode_injective = std::true_type;

        template <typename __InputRange, typename __OutputRange,
                typename __ErrorHandler>
        static constexpr auto encode(__InputRange&& __input, __OutputRange&& __output,
                state& __s, __ErrorHandler&& __error_handler);

        template <typename __InputRange, typename __OutputRange,
                typename __ErrorHandler>
        static constexpr auto decode(__InputRange&& __input, __OutputRange&& __output,
                state& __s, __ErrorHandler&& __error_handler);

};
```

3-speed approach

- Slowest Path (round-trip transcoding, non-eager)
- Faster Path (direct transcoding, no round-tripping, lazy)
- Fastest Path (bulk processing, eager consumption)

Slowest Path

- Works for Everything™
- Ideal for disparate encodings
  - SHIFT-JIS to GB18030
- Roundtrips through Unicode
- Converts one codepoint at a time



Slowest Path: Explicit Encode, Explicit Decode

- Roundtrip from encoding to decoding through common code point
- One-by-one encoding and checking
- Safe, scalable

```
__intermediate_code_point __intermediary_storage[8 * 1];
auto __scratch_space = ranges::span(__intermediary_storage, 8);
for (;;) {
    auto __decode_result = __encoding_from.decode(__working_input, __scratch_space,
    if (__decode_result.error_code ≠ encoding_errc::ok) {
        break;
    }
    auto __intermediary_storage_used = ranges::span(__intermediary_storage, __decode
    auto __encode_result             = __encoding_to.encode(__intermediary_storage_u
    if (__encode_result.error_code ≠ encoding_errc::ok) {
        break;
    }
    if (ranges::empty(__decode_result.input)) {
        break;
    }
    __working_input  = std::move(__decode_result.input);
    __working_output = std::move(__encode_result.output);
}
```

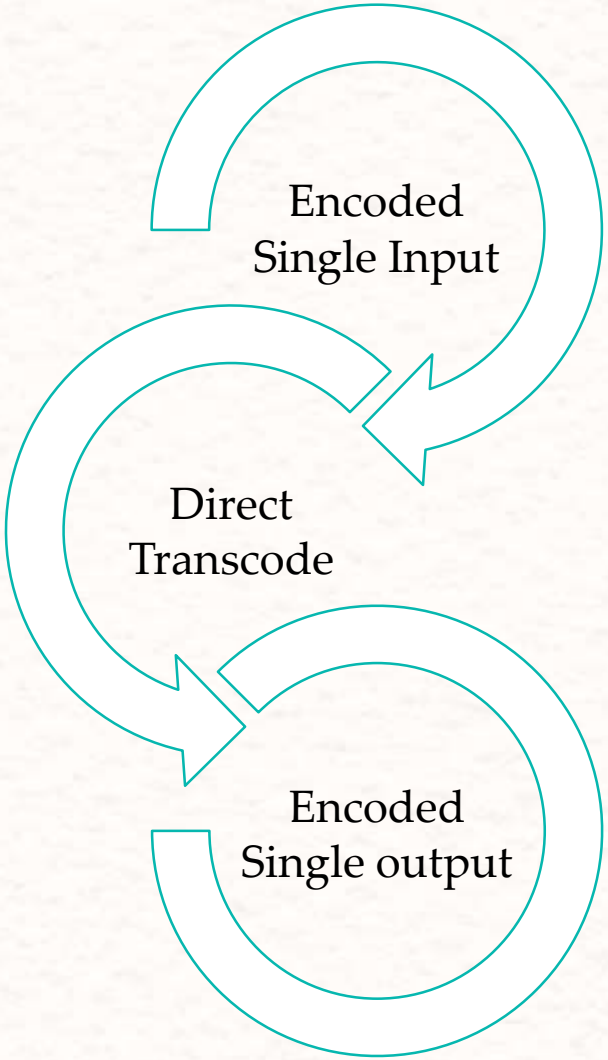
Slowest Path: lazy transcode_view 🐢

- transcode_view wraps a wrap a range
  - takes encoding objects as template parameters (From, To)
  - Round-trip converts

```
using namespace std::text;

transcode_view<std::u16string_view, utf16, utf32> lazy( u“🐶 woof!”);

for (const auto& utf32_cp : lazy_view) {
       char32_t u32_data = *utf32_cp;
       // do as you wish
}
```

Faster Path

- Specific to a direction and a pair of encodings
- Ideal for encodings where we know there are fast conversion (utf8 ←→ ut32, utf16 ←→ utf32)
- Converts directly without necessarily going to a code point
- One at a time conversion still



Faster Path: transcoding view/iterators 🐇🏃

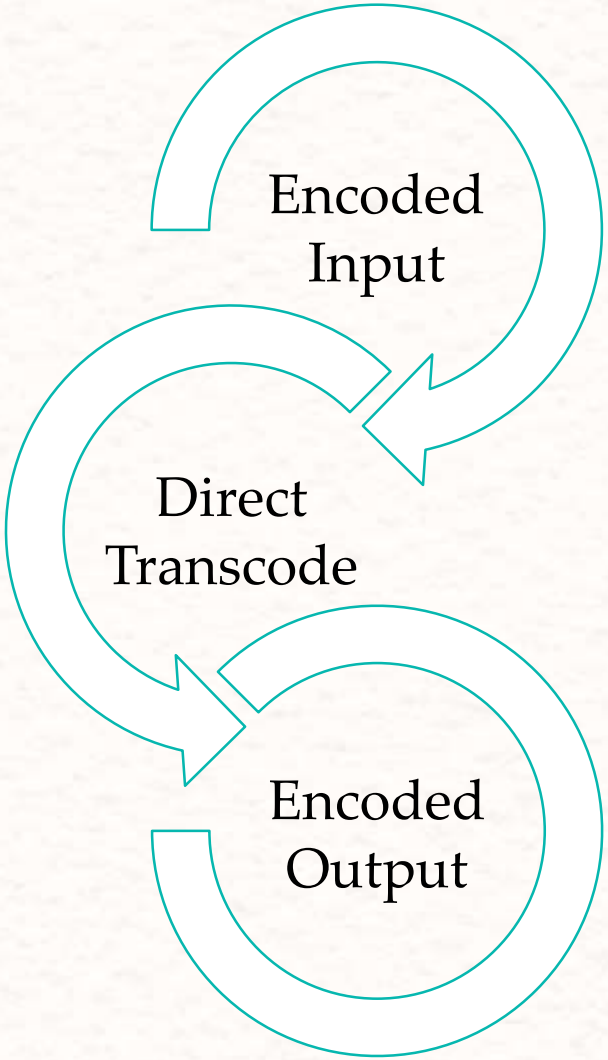
- Mostly invisible to user:
  - `transcode_view` customization points underneath do direct one-by-one conversion work
  - Speed up conversion by directly transcoding from one encoding scheme to another

- Faster, but not fastest: see Zach Laine's work in Boost.Text and June 26th SG16 discussion
  - https://github.com/sg16-unicode/sg16-meetings#june-26th-2019

```
using namespace std::text;

transcode_view<std::u16string_view, utf8, utf32> lazy( u8“🐶 woof!”);

for (const auto& utf32_cp : lazy_view) {
       char32_t u32_data = *utf32_cp;
       // do as you wish
}
```

Fastest Path

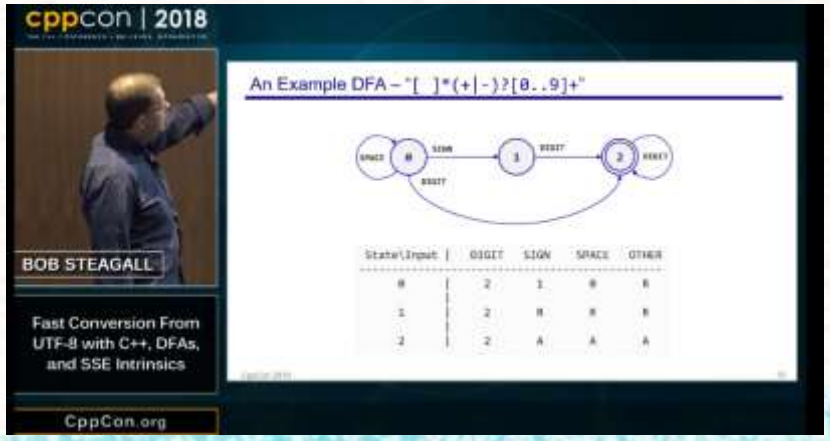
- Specific to a direction and a pair of encodings
- Bulk transcode converts directly without necessarily going to a code point
- Bulk conversion (SIMD, etc.)
- Typically done on `ContiguousRanges`



Fastest Path: eager function

- Absolute fastest path
  - Uses a free function to convert ***as much as possible***
  - Allows optimizations à la Bob Steagall: https://www.youtube.com/watch?v=5FQ87-Ecb-A

```
using namespace std::text;

u8text my_utf8_string = transcode<utf8>(U“ᛗᚣᚴ ᚴᚣᛋᛊ ᛁᛏᚣᚾ, ᚾᛁ ᛗᛁᛊ ᛃᚢ ᚾᛊᚣᚾ ᛒᚱᛁᚴᚴᛁᚿ.”);
```



Customization

- People outside the standard will have encodings and encoding conversion pairs they care about
  - Planned to have std::text::transcode, std::text::encode, and std::text::decode be customization points (Niebloids or Partial Struct Specializations)
  - Takes pressure of std implementers to have to provide amazing QoI at the very beginning

- Users can substitute in their proprietary converts for the 90% cases they care about
  - The rest will still work through the “Faster” and “Slower” paths

Implementation

- Work has begun
  - https://github.com/ThePhD/phd/tree/master/include/phd/text

- Need to go from working Proof of Concept to real implementation
  - Seeking grant to do work over the end of the Summer and for the Fall Semester as a Thesis Course

Join us online, at the mailing list, in our repositories, open teleconferences, and more:

https://github.com/sg16-unicode/sg16

Support my work:

https://github.com/users/ThePhD/sponsorship

https://thephd.github.io/support/